Java Programming Assignment
(2016 Term 2)

Introduction to Information Technology 4478

Note 1: Only use Java classes that are part of the standard Java SE SDK distribution! For the purpose of this assignment, you are not allowed to use any third party classes except for UCanAccess if you attempt stage 4.

Note 2: Familiarise yourself with the Java Style Guide at the end of this document.

All cases of plagiarism will be pursued! If in doubt, consult the Student Academic Integrity Policy https://guard.canberra.edu.au/policy/policy.php?pol_id=3175 or ask your lecturer.

The context for this assignment (all parts) is an examination of the growth function (also known as the exponential function) and inflation. This assignment will test a student's knowledge of and skills in writing application software for a particular task, understanding the business rules of a particular problem, coding these in a computer program, developing a graphical user interface, reading data from a text file on disk, and connecting to an SQL database from within the program. As such, the assignment requires you to integrate and synthesise what you have learnt so far, in order to design and create a correctly working solution. For this assignment, students will use the Java programming language and development will be on the Eclipse IDE platform as practised in the computer lab classes. This assignment consists of several stages.

- Stage 1: A simple console program (no GUI)
- Stage 2: The same but wrapped in a GUI
- Stage 3: Input comes from a text file – read only once, then information stored in a suitable data class, array, etc.
- Stage 4: Input comes from an SQL database file.
-

4478 IIT
Part A – Stage 1 (25 marks)
Part B – Stage 2 (10 marks)
Part C – Stage 3 (15 marks)
Bonus – Stage 4 (up to 10 marks)

Stage 4 is a bonus stage for undergraduate students to give you a stretch target. It allows you to pick up extra marks to make up for marks lost elsewhere in the assignment, but note that you cannot achieve more than 50/50 marks.

This assignment is to show your understanding of simple programming constructs; assignment statements, calculations, loops and arrays. For part A you will write a simple Java program to calculate the number of fish in a pond after a given number of generations, given a starting population and two different ways for the population to

grow. Note that the pond has a limited carrying capacity (5000 for part A), any fish spawning over that number will die due to overpopulation.

The first scenario is that the number of fish grows by a given percentage in each generation. For example, if the number of fish in the pond at the start is four and the growth rate is 50%, after one generation there are six fish (4*1.5 or 4*1+50/100), after two generations there are nine fish (6 * 1.5) and so on. Note that if the number of fish doubles in every generation, the growth rate is 100% or 1 + 100/100 = 2.

The second way for the fish population to grow is by a varying rate each generation, for ten generations.

Test Cases

When designing and implementing software applications, it is always important to first work out how to test the program(s) and what data to use, then to code. The following test cases have been designed to systematically test different conditions in the above rules. Use these to test your program, but make sure that your program can also handle other data, including negative growth in one or more generations.

Test Description	Option	Starting Population	Number of Generations	Growth Rate	Final Population	Died
Fixed rate (doubling); under pond capacity	F	100	5	100	3200	0
Fixed Growth (80% each generation); over pond capacity	F	200	6	80	6796	1796
Variable Growth; no growth;under pond capacity	V	10	10	0,0,0,0,0,0,0,0,0,0	10	0
Variable Growth, under pond capacity	V	70	10	15,20,25,30,50,70, 60,80,50,2	3746	0
Variable Growth; over pond capacity	V	100	10	100,100,80,70, 60,100,10,20,5,85	10038	5038

Submission Instruction

Add all Java files (ending in .java) to a ZIP file. You do not need to zip the entire project folder, just the Java source file(s) (the 'src' folder). Submit the ZIP file via Moodle (max. size 10MB). When ready for marking, make sure you click on "Submit the Submission".

There is no need for a cover sheet. By electronically submitting your assignment on Moodle, you acknowledge that this submission is your own work.

Stage 1: (25 marks).

Write a simple Java program to calculate the number of fish in a pond after a number of generations. Input the starting number of fish from the console, along with the growth rate (as a percentage) and the number of generations. The pond has a capacity of 5000 fish.

Section 1 (Fixed rate of growth)
The program must run in the terminal window. At a minimum, when your program is run, the terminal window must contain (with words that explain each number) the values of the starting population, the growth rate (as a percentage) the CAPACITY of the pool, the number of generations the program was run for, and whether or not there is still room in the pond. If the pond becomes full, also state how many fish died from over-population. You should run the program several times, changing the value of the starting population, the growth rate and the number of generations. On one run, the pool should still have some room (generations must be at least 5); on another it should exceed the pool's capacity.

Section 2 (Variable growth rates)
Run the program for 10 generations, but this time the growth rate of the fish population will vary from generation to generation. Enter the growth rate for each generation in the console and store the values in an array. Eg
100 100 80 70 60 100 10 20 5 85

With a starting population of 100 fish, and the growth rates above, the population after each generation will be:
200 400 720 1224 1958 3916 4307 5168 5426 10038

As the pond has a capacity of 5000, the number of fish that died due to overpopulation is 5038.

In Stage 1, you will be developing a Java program without a GUI. Input and output are via the console.

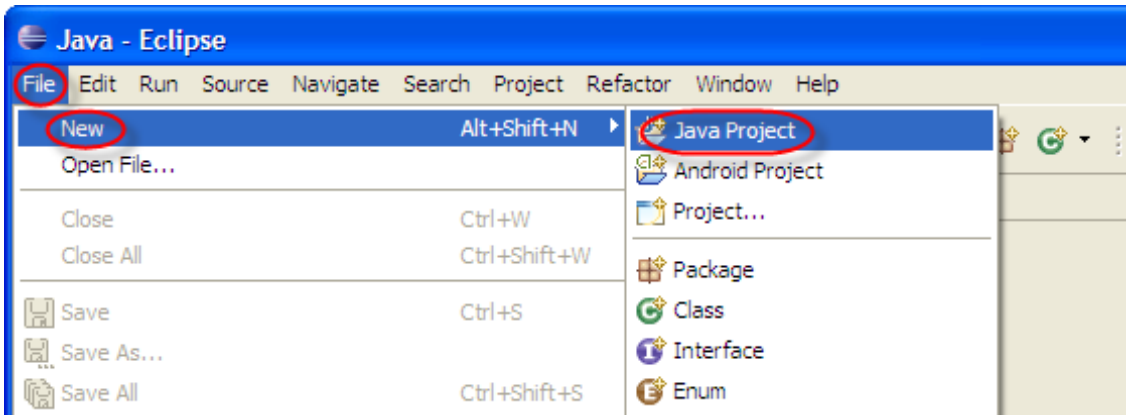
Step-by-Step Guide for Stage 1

1. Think about your strategy for how you will calculate the population after each generation. Is the growth rate fixed or variable? For a fixed growth rate (section 1), how many generations will the program run for? What is the final population? How many fish died? Hint: those over the capacity of the pond. For section 2, you

need different rates for each generation. Different strategies are possible. Here is one:

a. Use an integer array of 10 elements to store the different growth rates.
b. Once the values for the growth rates have been entered by the user via the console and stored in the array, multiply the current population by the growth rate and add the resulting number to the 'current' population, creating the new value for the population. Note that you can't have part of a fish!

2. Create a new Java project.



3. Add a simple Java class named `Stage1`. Do not add a GUI.

4. In the `Stage1` class (file `Stage1.java`), you may put all code for the user interaction and the calculation into the `main` method. (You might still need to define global variables and constants outside the main method at the start of the class.) Declare and instantiate the variables that you will need for section 1 (starting population, number of generations). For section 2 you will also need an integer array that will hold the growth rates, at the start of the main method, for example:

```
int[] iaGrowthRate = new int[10];
```

Add code to read in the starting population, then the ten values for each generation's growth rate from the console. In Java, there are different ways to do this. A recommended way is to use the Scanner class:

```
Scanner inConsole = new Scanner(System.in);

// Read the starting population
System.out.println("Enter the starting population of fish");
```

```
iStart = inConsole.nextInt();


// Ask whether the growth rate is fixed or variable
System.out.println("Enter F for fixed growth, V for
variable")
cOption = inConsole.next().charAt(0);


// Read the fixed growth rate
iRate = inConsole.nextInt();


// Variable Growth
// First growth rate for generation 1
System.out.print("Enter the growth rate for generation
one: ");
iaGrowthRate[0] = inConsole.nextInt();
```

Repeat (and adapt) the last two lines for the other growth rates for the next nine generations.

5. Now add the code that implements your strategy of calculating the final population of fish.
6. Finally, add two System.out.println() statements that state the final population and the number of fish that died due to overpopulation.
7. Test your implementation with the test cases mentioned above (and additionally your own).

What the tutors will be looking for

The tutor's instructions include

- Constants vs literals. Using constants is important for the ease of maintenance. Not using constants will result in lower marks. For example, consider constants for the default number of scores in each run or the minimum and maximum possible individual scores.
- Program code layout. Separate blocks of code by a blank line. Use comments.
- A comment is not an essay. Comments are important for the maintenance of a program and should contain enough details, but keep them concise. Do not comment every single line.
- The program must have a prologue. **Check style against the Java style guide attached below (last 4 pages).**

- Good names for your variables and constants. **Check style against the Java style guide attached below (last 4 pages).**
- Does the program work correctly?

Please refer to the Java Assignment Marking Feedback sheet for details on marks.

Stage 2

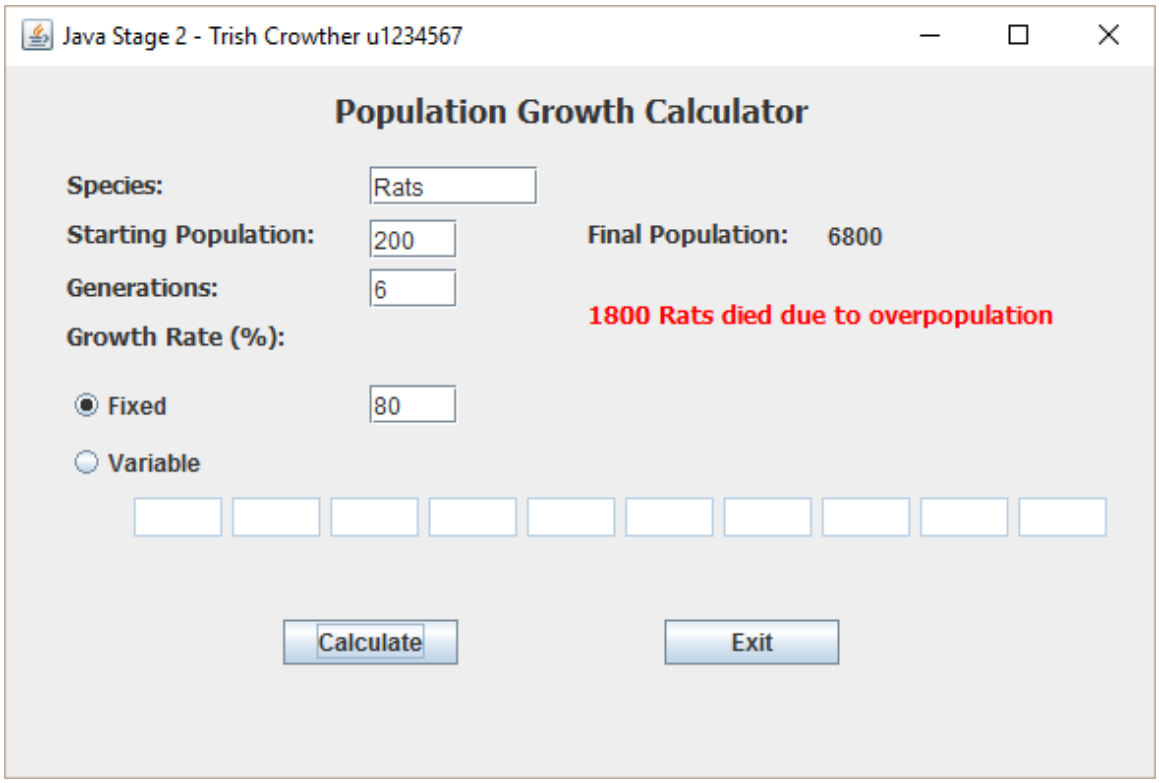
As the user input and program output via the console is not very satisfactory from a human-computer interaction and usability point of view, your task in this stage is to design and implement a Java Swing GUI application (using the built-in WindowBuilder in Eclipse and Java Swing components) that provides an easy to use interface.

This GUI application must allow a user to input the starting population, whether this is a fixed growth rate or a variable growth rate, and the growth rate(s). To this end, the user can

- input the data using Java Swing GUI elements, e.g. text fields, radio buttons, drop down lists, etc.,
- click on a *Calculate* button that starts the calculation of the final population from the current input and displays the output (including the number of creatures that died), and
- an *Exit* or *Quit* button to properly close the program.

Use the same code for the calculation of the scores as in Stage 1. Reusing code is an important part of software development.

You have a great degree of freedom in what GUI elements you choose and how you would like to design the layout of your GUI. The below example is really just that – an example the design, which you may copy if you are feeling uninspired to come up with your own design. What matters is the functionality of the design and that the user can input the required data in a sensible fashion.



Example of what the "empty" GUI might look like. You are free to use your own layout and other GUI elements, as long as the functionality is correct.

Notes:

- For this assignment, you do not have to do any checking for invalid user input (you may of course, if you want to), for example you do not have to check whether the user has typed in a negative number for the starting population or a letter in the text field for the growth rate(s). Checking for invalid user input will be discussed in lectures later in the term.
- Your user interface does not have to be identical to the one above. This is just an example of how the GUI could look.
- Your GUI should update the output label "Final Population" and another label to show the number of creatures that died (if any) in an appropriate manner with the calculated values based on the rules when the Calculate button has been clicked.

What the tutors will be looking for

The tutor's instructions include

- Constants vs literals. Using constants is important for the ease of maintenance. Not using constants will result in lower marks. For example, in addition to Stage 1, consider constants for the default width and height of the text fields so as to easily achieve a uniform look.
- GUI Design. Is it simple to use and easy to understand? Are all required components there?

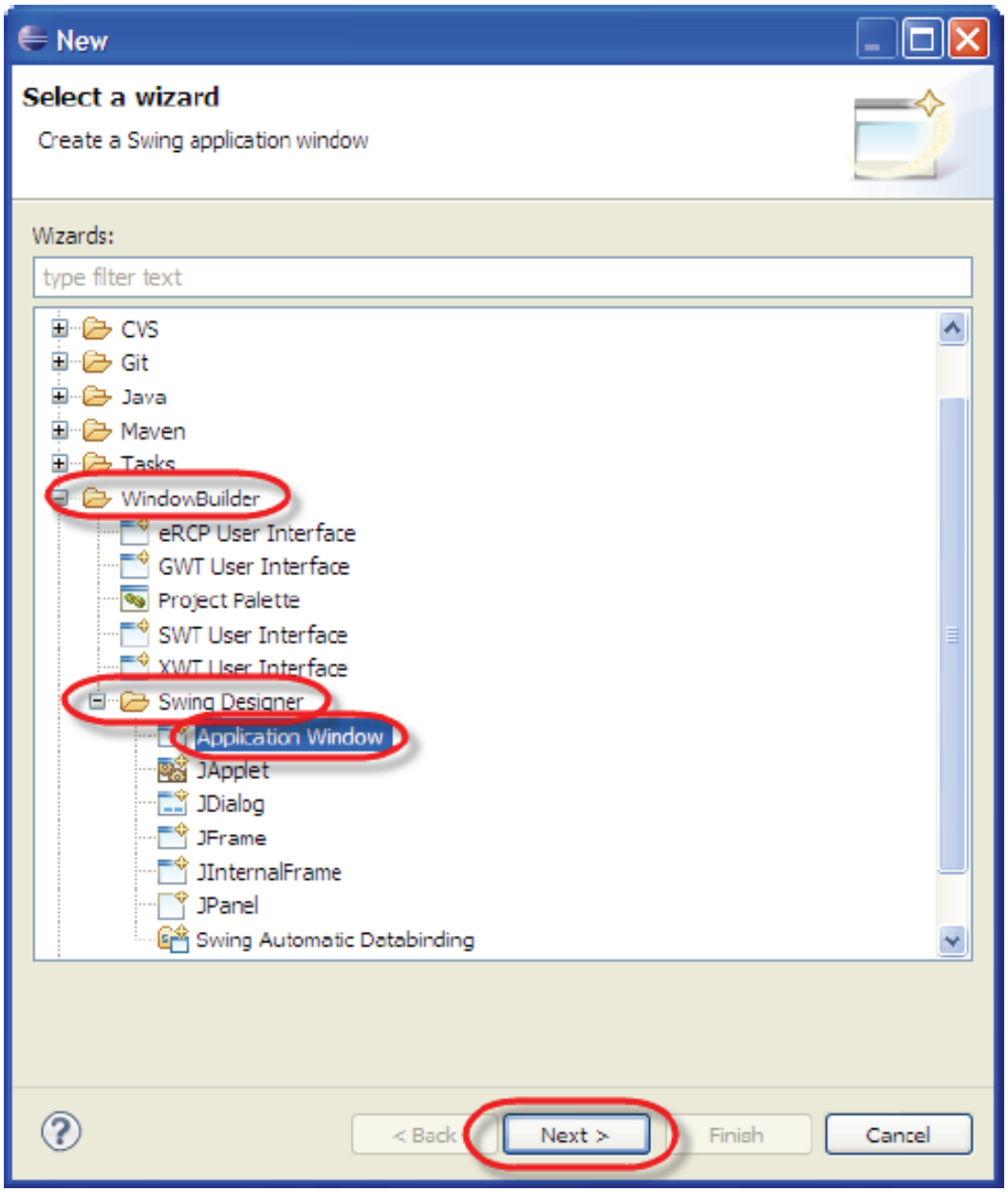
- Program code layout. Separate blocks of code by a blank line. Use comments.
- A comment is not an essay. Comments are important for the maintenance of a program and should contain enough details, but keep them concise. Don't comment every single line.
- The program must have a prologue. **Check style against the Java style guide attached below (pp. last 4 pages).**
- Good names for your variables and constants. **Check style against the Java style guide attached below (last 4 pages).**
- Does the program work correctly? Are the elements drawn the right way?

Please refer to the Java Assignment Marking Feedback sheet for details on marks.

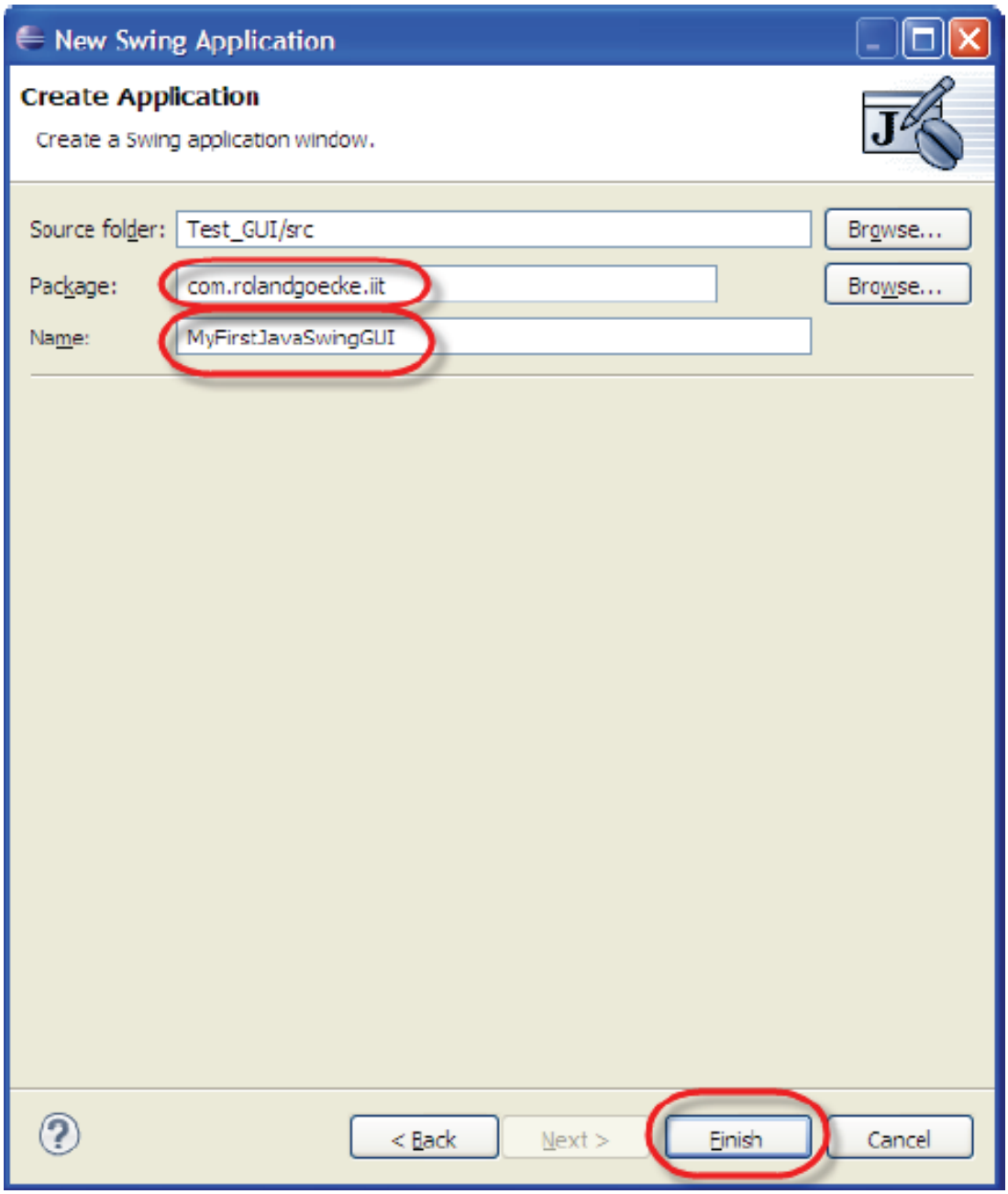
Step by Step Guide for Stage 2

1. Draw a sketch of your graphical user interface on paper. What Java Swing components will you use? Where will you place them? The above GUI is just an example. You can copy the design of it or create your own.

2. Add to the same **project** that was used for Stage 1, a ***new Application Window*** class (New → Other → WindowBuilder → Swing Designer → Application Window) for your GUI.
   (*See below for a screenshot.*)

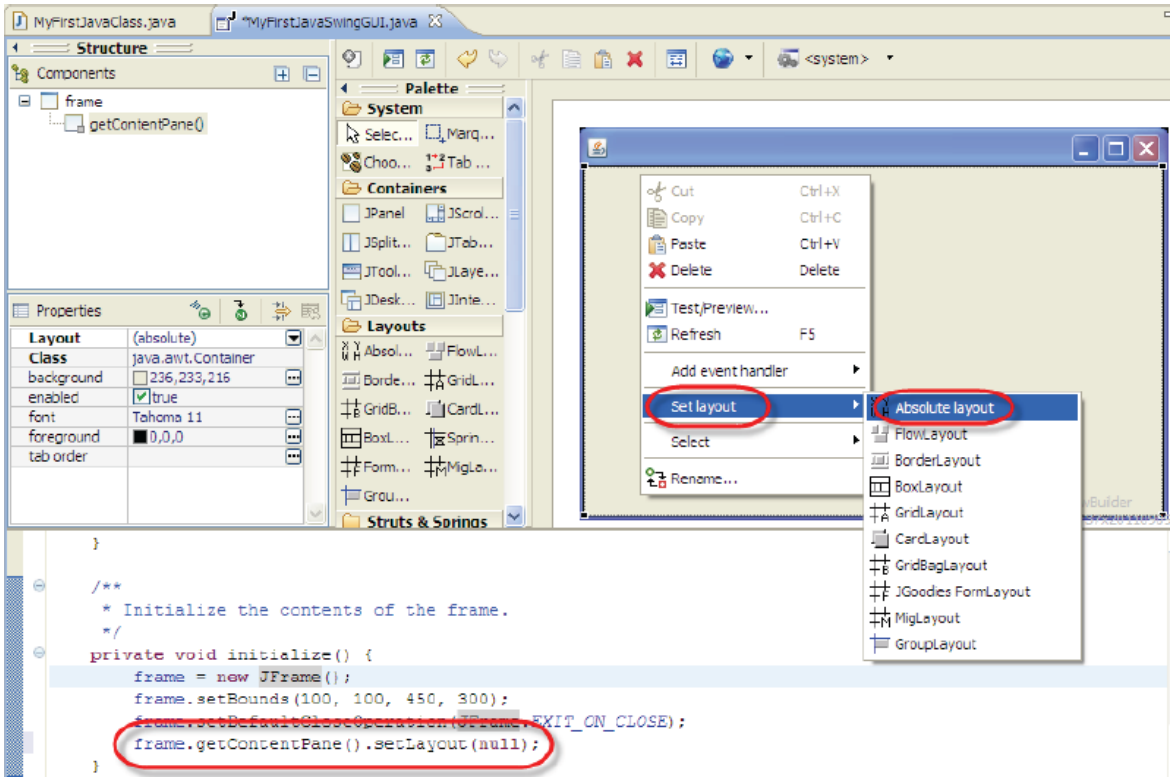


3. As in the lectures, workshops and lab classes, give that Java application a suitable ***name*** - Stage2 - (and optionally ***package name***) of your choice.



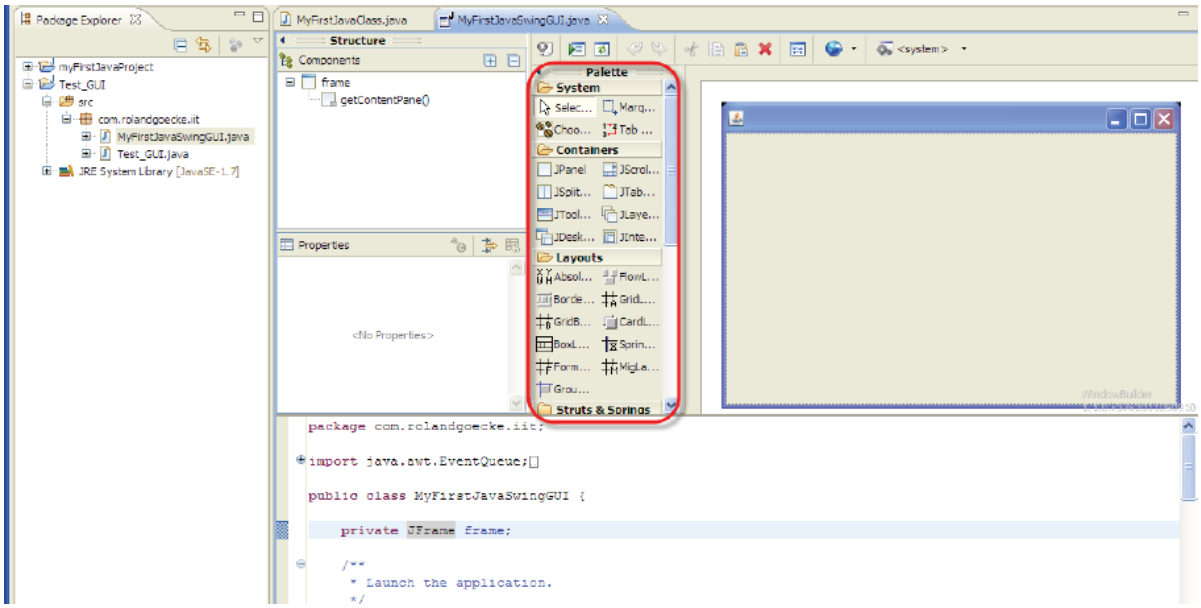
A new JFrame object will be created and you should be able to switch between the Java source code for it as well as the empty GUI in the WindowBuilder editor in Eclipse.

4. Right-click on the grey area inside the JFrame and set the layout to “Absolute layout”. (Note, this shows up as “null” layout in the source code.)

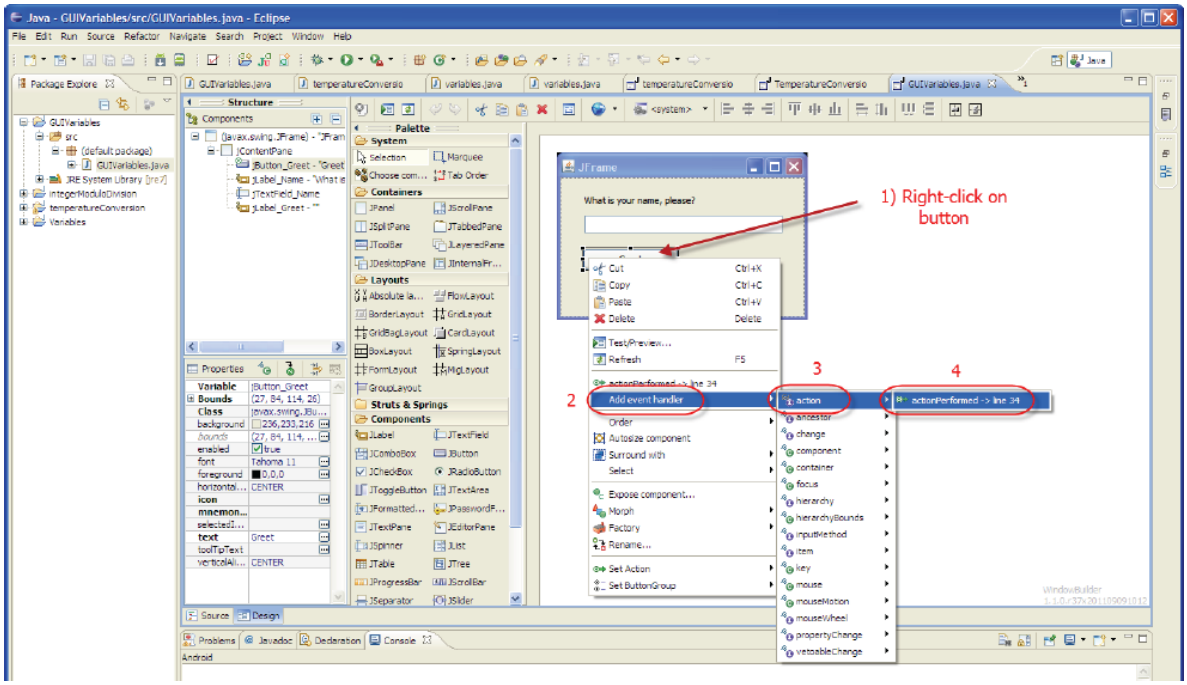


5. Adjust the size of the JFrame to a suitable size for your GUI.

6. Add all the components for your GUI. Use Java Swing components, which you can add via the Palette in the WindowBuilder’s Design view. Make sure that the names you use for these components comply with our Java Style Guide (see below).



7. Add event handlers (ActionListeners) for your buttons, radio buttons, check boxes, etc. To do so, in the GUI editor right-click on the element (say, a JButton) and select Add event handler → action → actionPerformed. (Similar events may be needed for your checkboxes, radio buttons, etc.)



8. Add the code that does the actions, e.g. that does the calculation of the final population and the number of creatures that died.

9. Reuse your code for the calculation Stage 1 by copying and pasting it from the Stage 1 main method into the Calculate button's event handler.

10. Test your application! Run it as a Java Application. Enter the test cases listed above and check if your program gives the same result.

11. Make sure your code adheres to the Java style guide. Check for prologue comment and other comments in your code. You need to manually add more comments than the automatically generated comments!

Stage 3

In Stage 3 of the assignment, the input will not come directly from the user any more, but rather from a text file.

The file growth.txt contains the growth rates of several species & their habitat. Each line of this file contains the growth rates of one species. Each line consists of the species name, habitat, starting population and growth rates, separated by semicolons. The growth rates for each generation consist of a comma-separated list of values. The following is an example:

```
Rats;Shed;6;15,20,25,30,50,70,60,80,50,2
```

You will need to copy `growth.txt` into the *same* directory of your Eclipse project that holds the .project file. When marking, the details in the file may change, but the structure will not.

Your task will be to add code to your program that reads the growth rate data from the text file, puts these into appropriate storage in memory (I suggest you create a data class similar to the example in star4.java and then create an array or vector that will hold each instantiation of that data class), adds the species names to a JList object, and allows the user to choose a species from this list.

Upon this selection, the program displays the species habitat for the currently chosen species in a text label (JLabel), displays the growth rates, computes the total population for that species after ten generations (without deaths) and how many died (if any), then updates a drawing panel to display the growth rates in `a bar graph like comparison`. These are **new components in Stage 3** that you can add to your Stage 2 GUI or write it as a separate GUI application (either way is fine).

Notes:

- For this part of the Java assignment, add Java Swing GUI components that allow the user to select a species name, via a JList, and that calculate the final population as well as draws a bar graph like comparison of the growth rates for a species in the text file.

- The bar graph should be drawn as soon as a species name is selected in the JList. It should show the growth rate for each generation as a vertical bar starting from a common baseline near the bottom of the drawing area.
- Update the "Final Population…" and "… died due to overpopulation" labels according to the information for the currently selected species by getting the relevant information from the storage in memory (e.g. the data class) and calculating the population and casualties (number that died).
- The average growth rate for the species should be shown as a coloured horizontal line across the graph. Choose a suitable colour that is different from the colour you used for the vertical bars representing the growth rates for each generation.
- Try to reuse code, such as the code for the drawing, from Stage 1 and Stage 2. This will mean that you should have the GUI code separated from the drawing code and the file reading code (i.e. in a separate method or separate methods). This is good programming style!
- Remember that it is good practice to have a *Quit* button in your GUI to exit the program.
- The drawing area must have a minimum size of 300 x 200 pixels (but can be bigger). The drawing area should have a clearly visible border, but the bars / rectangles and lines not.

What the tutors will be looking for

The tutor's instructions include (apart from the usual such as good variable names, prologue / comments, code layout, …)

- Use of constants. Think about where you could use constants here. For example, consider the width of the bar graphs to be a constant.
- The text file should only be read once. You probably want to do that at the start of the program. Consider how you would do that.
- Correct display of species information based on the details in the file.
- Separate GUI functionality from drawing. Students should use a separate method for the drawing. It is good practice and follows good programming style to separate GUI code from other code.
- There are various ways you can store the data when reading the file. You could use a multidimensional array, or create your own data class and then use an array or vector of this new data type. Using your own data class is preferential.

Stage 4

Here, the input will now come from an SQL database contained in the MS Access file `growth.mdb`. This file is available on the IIT / IIT G site on Moodle (UC LearnOnline).

There is one table in the database, *tblSpecies*.

The table *tblSpecies* contains fields “speciesName”, “habitat”, “startingPoplution”,“Gen1_Growth”, “Gen2_Growth”, “Gen3_Growth”, “Gen4_Growth”, “Gen5_Growth”, “Gen6_Growth”, “Gen7_Growth”, “Gen8_Growth”, “Gen9_Growth”, “Gen10_Growth”.
A typical record would be < `Fish,Pond,5,100,100,80,70,60,100,10,20,5,85`>.

Write a Java program with a GUI (Note: extending your GUI program from Stage 3 is acceptable, but you could also write a separate Java file for Stage 4; either way is OK) that allows the user to do the following:

- Let the user select the name of a species (e.g. from a JList) and display the species growth rates in the drawing area as in Stage 3. Update the labels in the GUI showing the growth rates, species name, habitat, etc.
- Create a database report consisting of all species’ growth rates, starting population, final population and deaths, and present the report with the following columns: “Species”, “Habitat”, “Growth Rates”, “Starting Population”, “Final Population” and “Deaths”. The report should be in ascending alphabetical order of “Species”. It should be written to a file “`database_report.txt`”. Make sure that your columns are properly aligned. Text columns should be left aligned, number columns right aligned.

Notes:

- You may extend your GUI from Stage 3 to include Stage 4, in fact you could start from Stage 2. If you do, label the parts clearly with “Stage 2”, “Stage 3”, and “Stage 4”, so that your tutors know which part of your program is in reply to what part of the assignment. Alternatively, you may write a separate file for each stage but can reuse code.
- Use a *Disconnected Database Access* model, i.e. connect to the database, run the SQL command, get the resulting virtual table in the `ResultSet` object, then disconnect again. Do not connect to the database and store all database information in local storage.

Database Report for Stage 4

Each database report should

- include a header,
- include a column header,
- have columns lined up, with text columns left justified and numeric columns right justified.
- make provision for multi-page reports. At least one of your reports should actually be more than one (fictional) page in length. Do this by defining the report page length as 5 lines.

What the tutors will be looking for

The tutor's instructions include (apart from the usual such as good variable names, prologue / comments, code layout, …)

- Correct connection to the database.
- Correct identification of all athletes' scores. Show each judge's score in a separate text label.
- Correct visualisation of the scores as a bar graph for the selected athlete in the drawing area as in Part A.
- Make sure that the lines in the database reports are split into pages and the columns are lined up and correctly formatted.

Java Style Guide

General

Your programs should be

- Simple
- Easy to read and understand
- Well structured
- Easy to maintain

Simple programs are just that. Avoid convoluted logic, nested if-statements and loops, duplicating execution (such as reading files multiple times), repeated code, being "clever".

Programs can be made easier to read by following the "Layout" and "Comments" guidelines below.

Well structured code uses methods and functions to help tame complexity.

If programs are simple, easy to understand and well structured they will be easy to maintain. Easily maintained programs will also use constants rather than literals, and use built-in functions and types.

Layout

The text editor in the Eclipse IDE does a good job of automatically laying out your program. You can control this operation to some extent (using the Tools/Options menu). However, you are unlikely to need to do so.

White space

Use white space freely:

- A blank line after declarations
- 2 blank lines before each function declaration
- A bank line between sections of a program or after a sequence of statements to separate a block of statements.

Names

Well chosen names are one of the most important ways of making programs readable. The name chosen should describe the purpose of the identifier. It should be neither too short nor too long. As a guide, you should rarely need to concatenate more than 3 words.

Letter case

Constants use ALL_CAPITALS.
e.g. PI, MAXSTARS

Variables use firstWordLowerCaseWithInternalWordsCapitalised
e.g. cost, numStars

It is a good idea to use the so called ***Hungarian notation*** where the first letter (or two) indicate what type the variable has:

- i for int, e.g. iNum
- l for long, e.g. lNum
- c for char, e.g. cLetter
- b for byte, e.g. bNum
- f for float, e.g. fFloatingPointNum
- d for double, e.g. dFloatingPointNum
- s for String, e.g. sName
- bo for Boolean, e.g. boCar

Methods (subs and *functions)* use firstWordLowerCaseWithInternalWordsCapitalised e.g. cost(), numStars()

Recall that the IDE preserves the capitalisation of the declarations.

Types of names

1. Names of *controls / GUI elements* should have a prefix indicating the type of control (Java Swing). The prefix should be lower case with the remainder of the name starting with upper case.

Control	prefix	example
text field	`jTextField`	`jTextField_Quantity`
text box	`jTextArea`	`jTextArea_Amount`
label	`jLabel`	`jLabel_Result`
list box	`jList`	`jList_Students`
combo box	`jComboBox`	`jComboBox_Shares`
command button	`jButton`	`jButton_Quit`
radio button	`jRadioButton`	`jRadioButton_Colour`
check box	`jCheckBox`	`jCheckBox_Salt`
panel	`jPanel`	`jPanel_Drawing`
track bar / slider	`jSlider`	`jSlider_X`

2. Names of *functions* should be **nouns** or **noun-phrases**

   Example:

   ```
   newSum = sum(alpha, beta, gamma)
   ```

   or

   ```
   newSum = sumOfStudents(alpha, beta, gamma)
   ```

   rather than

   ```
   newSum = ~~computeSum(alpha, beta, gamma)~~
   ```

3. Names of *methods* should be **imperative verbs**

Example:

```
printResults(newSum, alpha, beta)
```

rather than

~~`results(newSum, alpha, beta, gamma)`~~

4. Names of *Boolean functions* should be **adjectives** or **adjectival phrases**

   Example:

   ```
   if (empty(list))
   ```

   or

   ```
   if (isEmpty(list))
   ```

   rather than

   ~~`if (checkIfEmpty(list))`~~

Comments

Comments should explain as clearly as possible what is happening. They should summarise what the code does rather than translate line by line. Do not over-comment.

In Java, you can either use `/* Comment */` for a block comment possibly extending over multiple lines, or you can use `// Comment` for a one-line comment.

Comments are used in three different ways in a program:

- Heading comments
- Block comments
- End-of-line comments.

Heading comments: These are generally used as a prologue at the beginning of each program, procedure and function. They act as an introduction, and also describe any assumptions and limitations. They should show the names of any files, give a short description of the purpose of the program, and must include information about the author and dates written / modified.

Block comments: In general, you should not need to comment blocks of code. When necessary, these are used to describe a small section of following code. They should be indented with the program structure to which they refer. In this way the program structure will not be lost.

End-of-line comments: These need to be quite short. They are generally used with parameters to functions (to explain the purpose and mode of the parameter), and / or with variable declarations (to explain the purpose of the variable), and are not meant to be used for code. In Java, use `//` for end-of-line comments.

Comment data, not code: Comments about the data – what it is and how it is structured - are much more valuable than comments about the code.

Prologue

Each of your programs should have a prologue. This is a set of "header comments" at the top of Form 1. It should include

who wrote it	`// Author: Roland Goecke`
when it was written	`// Date created: 12 Feb 2013`
when it was last changed	`// Date last changed: 1 Mar 2013`
what it does	`// This program does...`
what files it reads / writes	`// Input: sample.txt, Output: none`

Constants

A literal is a number such as 10 that is stored in a variable. You should avoid literals in your program code. Instead use constants. They are stored differently in the computer memory and can be used more efficiently by the compiler / interpreter. For example, instead of

```
int iFine = 80;
```

use

```
public static final int SPEEDING_FINE = 80;
...
iFine = SPEEDING_FINE;
```

This makes your program easier to maintain. If the speeding fine changes, the change only has to be made in the one place.

Methods and Functions

Using methods and functions is one of the simplest and most effective ways of dealing with complexity in a program.

> When you write your own method or function to perform a calculation, it should not refer to any GUI controls.
> Do not mix IO and calculations in a method or function.

Method

- encapsulates a task
- name should be a verb. Example**: printReport**()
- should only do 1 task.

Function

- encapsulates a query
- return type is **int, boolean, double**…;
- if return type is **boolean**, name should be an adjective, otherwise name should be a noun.
  - Examples: **isEmpty(list), SquareRoot(number)**

- should answer only 1 query.

Comments for each function and method (to be placed on the line before the method / function starts)

name	above rules apply
purpose	what it evaluates or what it does
assumptions	any assumptions made, particularly on the arguments

Duplicated Code

View any duplicated code with suspicion. Look for a way of factoring the duplicated code into a method.

Built-in functions

Unless you have a good reason, use built-in functions rather than writing (or not writing) your own. They will be correct, flexible and familiar to a reader.

Types

Using types is an important way of making your intentions clear. Java does not allow you to be sloppy with your types. In IIT / IIT G, you are expected to choose appropriate types. You are also expected to use type conversions such as `Integer.parseInt("123")` and `Double.toString(123.45)`.

Simplicity

Simplicity has long been recognised as one of the most important characteristics of programming. There are many ways of simplifying programs. These include avoiding nested IF statements, nested loops and complex conditions.